Embedded Systems Programming

Internet Appliance Design

An Introduction to USB Development

Jack G. Ganssle

USB is supplanting old-fashioned parallel and serial interfaces in all sorts of applications. This article will get you well on your way toward including USB support in your product.

The Universal Serial Bus (USB) was born out of the frustration PC users experience trying to connect an incredibly diverse range of peripherals to their computers. It's the child of vendors whose laptops require a small profile peripheral connector. It further promises to reduce the proliferation of cables and wall transformers that overwhelm even the smallest computer installation.

Above all, USB offers users simple connectivity. It eliminates the vast mix of different connectors for printers, keyboards, mice, and other peripherals. In a USB environment, DIP switches aren't necessary for setting peripheral addresses and IRQs. It supports all kinds of data, from slow mouse inputs to digitized audio and compressed video.

Perhaps USB seems an inappropriate topic for *Embedded Systems Programming.* Why would embedded folks care about something as PC-centric as USB? The fact is, every USB device is an embedded system. If you're building an application that connects to a PC, realize that USB is supplanting old-fashioned parallel and serial interfaces. Sooner or later you'll have to migrate away from RS-232 since there are other options (USB, Firewire, and others).

USB overview

USB is a serial protocol and physical link, which transmits all data differentially on a single pair of wires. Another pair provides power to downstream peripherals.

The USB standard specifies two kinds of cables and two variations of connectors. High-speed cables, for 12Mbps communication, are better shielded than their less expensive 1.5Mbps counterparts. Each cable has an "A" connector on one end and a "B" on the other. Figure 1 shows that "A" connectors go to the upstream connection while the "B" version attaches downstream. Since the two types are physically different it's impossible to install a cable incorrectly.

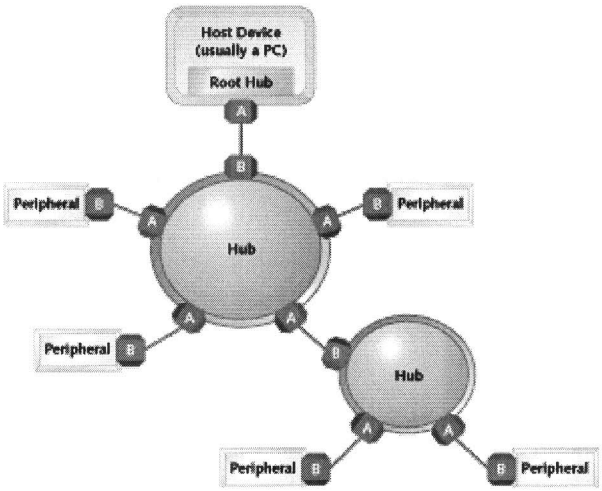

FIGURE 1: The USB "tiered star" topology

Power management

"Dad, you've got to turn that laptop off before connecting the disk drive." My 12-year-old understands the peril of connecting a floppy to a running computer. But this drive's USB link makes such worries obsolete. I'm free to install and remove this USB peripheral regardless of power state.

Two of the four wires in a USB cable supply power to peripherals. Though nominally +5V, the spec allows for quite a bit of variation in this; designers should allow for as little as about 4V. A peripheral that draws up to 100ma can extract all of its power from the bus wiring all of the time. Higher current requirements are trickier; if the device requires less than 500ma, and if the upstream host or hub can provide that much power (which is optional), the device can be bus-powered if at power-up time, during system configuration, it consumes less than

100ma. If the device needs more than a half-amp, then it must have its own power supply.

USB hosts and hubs manage power by enabling and disabling power to individual devices to electrically remove ill-behaved peripherals from the system. Further, they can instruct devices to enter the suspend state, which reduces maximum power consumption to 500 microamps (for low-power, 1.5Mbps peripherals) or 2.5ma for 12Mbps devices.

These are average states over a one-second period. A low-duty cycle event can consume more power as long as it meets the average spec.

Cable lengths are limited to 5m for 12Mbps connections and 3m for 1.5Mbps. This sounds backwards, but stems from the use of better cables at higher speeds. USB's topology is a "tiered star" (see Figure 1). Hubs are the communication nodes that interconnect devices. One, and only one, "host" device (typically a PC) includes the "root hub" which forms the nexus for all device connections. The host is the USB system's master, and as such, controls and schedules all communications activities.

Peripherals, the devices controlled by USB, are slaves responding to commands from the host. When a peripheral is attached to the USB network, the host communicates with the device to learn its identity and to discover which device driver is required (a process called **enumeration**). To avoid the DIP switch and IRQ headaches of the past, the host supplies a device address to the peripheral during enumeration.

The specification recognizes two kinds of peripherals: stand-alone (single function units, like a mouse) or compound devices (those that have more than one peripheral sharing a USB port). An example of a compound device is a video camera with separate audio processor.

Hubs are bridges; they increase the logical and physical fan-out of the network. A hub has a single upstream connection (that going to the root hub, or the next hub closer to the root), and one to many downstream connections.

Hubs are themselves USB devices, and may incorporate some amount of intelligence. A critical part of the philosophy of USB is that users may connect and remove peripherals without powering the entire system down. Hubs detect these topology changes. They also source power to the USB network; power can come from the hub

itself (if it has a built-in power supply), or can be passed through from an upstream hub.

Once a hub detects a new peripheral (or the removal of one), reports the new information to the host, and enables communications with the newly inserted device, it essentially becomes invisibleýa smart wire, passing data between the host and the devices. Though physically configured as a tiered star, logically (to the application code) a direct connection exists between the host and each device.

USB comm overview

USB communications takes place between the host and **endpoints** located in the peripherals. An endpoint is a uniquely addressable portion of the peripheral that is the source or receiver of data. Four bits define the device's endpoint address; codes also indicate transfer direction and whether the transaction is a "control" transfer. Endpoint 0 is reserved for control transfers, leaving up to 15 bi-directional destinations or sources of data within each device.

The idea of endpoints leads to an important concept in USB transactions, that of the **pipe** . All transfers occur through virtual pipes that connect the peripheral's endpoints with the host. When establishing communications with the peripheral, each endpoint returns a **descriptor** , a data structure that tells the host about the endpoint's configuration and expectations. Descriptors include transfer type, max size of data packets, perhaps the interval for data transfers, and in some cases, the bandwidth needed. Given this data, the host establishes connections to the endpoints through virtual pipes, which even have a size (bandwidth), to make them analogous to household plumbing.

USB supports four data transfer types: control, isochronous, bulk, and interrupt.

Control transfers exchange configuration, setup, and command information between the device and the host. CRCs check the data and initiate retransmissions when needed to guarantee the correctness of these packets.

Bulk transfers move large amounts of data when timely delivery isn't critical. Typical applications include printers and scanners. Bulk transfers are fillers, claiming unused USB bandwidth when nothing more important is going on. CRCs protect these packets.

Interrupt transfers, though not interrupts in the CPU-diverting sense, poll devices to see if they need service. Peripherals exchanging small amounts of data that need

immediate attention (such as mice and keyboards) use interrupt transfers. Error checking validates the data.

Finally, **isochronous** transfers handle streaming data like that from an audio or video device. It is time sensitive information so, within limitations, it has guaranteed access to the USB bus. No error checking occurs so the system must tolerate occasional scrambled bytes.

The host device driver

The days of simple interfaces like RS-232 are long gone, as are the frustrations of making incompatible devices talk reliably. USB is a complex standard that requires an enormous amount of software support, both on the firmware side and in the host computer.

Most host-end connections, for better or worse, will be PCs running a Microsoft operating system. USB is not supported at all in DOS, Windows 3.x, or Windows NT. Windows 95 provided some USB drivers, though only in the later versions starting with OEM Software Release 2.1. All Windows 98 releases include a full set of drivers for common USB applications and Windows 2000 (the next generation of both 98 and NT) will too.

Some of the most brilliant firmware engineers quail at the thought of writing Windows drivers, with good reasons. Unhappily, a USB driver is a difficult beast. The good news is that in many cases the drivers provided with Windows will handle even your custom peripheral. Let's look at how Windows drivers function.

Microsoft's roadmap for drivers in Windows 98 and beyond relies on the Win 32 Driver Model (WDM), which layers different parts of the communications process into a stack of drivers (see Figure 2). Application code (via Windows API calls) communicates with class or custom drivers in the WDM. Within the WDM stack itself data transfers use lower-level IRP (I/O request packets) rather than API calls.

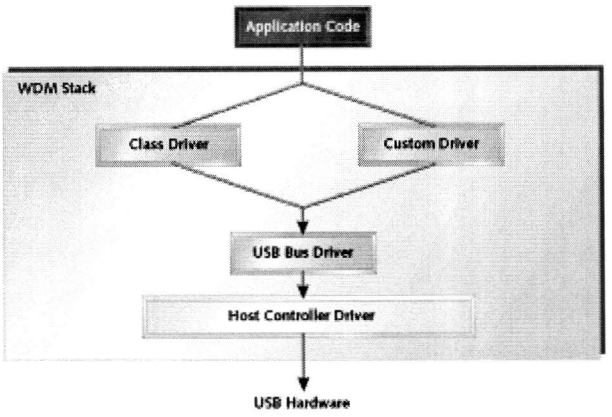

FIGURE 2: Windows USB WDMMode

The low-level USB bus driver manages USB device power, enumeration, and various USB transactions. Below this, the host controller driver talks directly to the USB hardware in the PC. Both of these drivers are supplied with current Windows versions; you won't have to write or modify either.

Windows, as well as the USB specification, segments drivers into "classes," where hardware that falls into a single class shares similar interfaces. A class defines a baseline specification for a given set of capabilities; all devices in a class require comparable types of software support.

An example is the human interface device (HID) class, which supports devices like mice, joysticks, and keyboards. Another is the monitor class, which controls image position, size, and alignment on video displays. Windows 98 ships with a complete HID class driver, so if your peripheral requires HID-like support, you may be able to use this built-in driver without writing any host code.

Current specifications of class drivers may be found on the USB home page (www.usb.org); Windows support for classes is available from Microsoft, but at this time is somewhat limited (though HID-class devices are indeed fully supported).

Custom drivers are an alternative to class drivers. A custom driver exploits the capabilities of a particular piece of hardware at the end of the USB cable. If you've built a data acquisition system, for example, odds are there's no class driver available so you'll have to write your own. Similarly, if your device has capabilities well beyond that of a standard class you may also have to write a custom driver to support these features.

Visual C++ can compile WDM drivers, of course. Download the Windows 98 Driver Developer's Kit (DDK) from www.microsoft.com/DDK/ddk98.htm, as this resource includes example code for several USB drivers.

BlueWater Systems (www.bluewatersystems.com) also has a driver development kit whose wizard greatly eases any sort of Windows driver development. An add-on, the USB Extensions Toolkit, is a boon for USB drivers, but be aware of the per-unit royalty charge.

Unless you're building a typically PC-centric peripheral like a mouse, you'll likely also create a host application that exchanges data with the USB device and interacts with the user. An oscilloscope, say, using an A/D converter and triggering logic located at the end of a USB cable, requires an application with a scope-like GUI. To exchange data with the USB device the application code simply issues standard file-like API calls, using a standard Windows handle to identify the device.

The chips

Since USB is (for all practical purposes) tied to the high-volume PC business, dozens of vendors offer hundreds of different support chips. The best reference to these ICs is IBH Doran's (a German consultancy company) Web site at www.ibhdoran.com/usb_link.html.

USB parts are rather hard to categorize, but fall generally into three camps: host-side USB controllers (which live inside the PC, and are probably of little interest to ESP readers), devices designed as stand-alone USB peripheral controllers (like a smart UART, these chips handle communications but you'll need another microprocessor as the brains of your device), and versions of popular processors that include a USB interface. Using the UART metaphor again, this last group is like the high-integration CPU with an on-board UART; both your application code and that needed for USB control runs on the same part.

Beyond these three categories, some vendors offer specialized parts, such as USB camera controllers, audio devices, bridges that link USB to other buses, and specialized HID controllers.

It's impossible to do justice to various products here. A few highlights follow.

Cypress Semiconductor (www.cypress.com) has a variety of high- and low-speed chips based on 8-bit RISC cores with instruction sets optimized for USB applications. Both one-time programmable and EPROM parts are available. Their development kits (the starter

kit costs $99, but the much more useful developer's kit runs a still-reasonable $495) are the way to go for getting firmware running.

Cypress bought Anchor (www.anchorchips.com) last year; Anchor's EZ-USB 8051-based chips use a standard instruction set and come in a wide variety of RAM and ROM sizes. They, too, offer a $495 developer's kit.

Scanlogic's SL16-USB controller (www.scanlogic.com) is a custom-architecture 12Mbps controller. Their development board, like most of those offered by other vendors, includes WDM drivers. Interestingly, Scanlogic claims users can bring an embedded USB app up in only five weeks (on their hardware, of course).

Philips' PDIUSBD11 is an intriguing chip that connects a USB port to I^2C. I^2C is a speedy, two-wire serial interface that a lot of embedded systems already employ, and one that comes built into some microcontrollers. So, using the PDIUSBD11 you could conveniently tie your current I^2C-aware product to a PC's USB port.

A number of vendors offer versions of their controllers with an on-board USB port, giving you USB access without using a processor devoted to communications alone. Motorola's products range from a USB-aware 6805 (MC68HC05JB4) to the PowerPC MPC850. AMD added USB to the venerable x86 line with their 186CC. Both Atmel (AT43USB321) and Microchip (PIC 16C745) have microcontroller products with the communications link.

Some vendors offer low-level USB drivers you can embed into your products. Phoenix (www.phoenix.com/platform/usbaccess.html), building on their BIOS products, offers firmware-side USB stacks that tie into commercial real-time operating systems, such as those from Accelerated Technology, Lynx, and Integrated Systems/Wind River Systems.

Development tools are as important as chips and code. USB is a complex protocol that tosses a lot of data around. Debugging by looking at the serial stream on a scope is not efficient. Several companies offer protocol analyzers that monitor the USB link and display transmitted data in an understandable form. Two are Hitex's USB Agent (www.hitex.de/usb/protan.htm) and CATC's USB Chief Bus & Protocol Analyzer (www.catc.com/home.html).

USB in the lab

A lot of us use PCs to control short-run products, or to handle simple monitoring tasks in the lab. RS-232 and parallel printer ports, the staple connection for many of

these applications, are often just not available on recent PCs. In fact, many laptops now offer these only on an expansion port, yet include USB on the main unit.

Several companies now sell data acquisition products that incorporate a USB link. For example, National Instruments, the people who provide the popular LabView software package, sells the "DAQPad" family of instruments (www.ni.com/daq/10usbdaq.htm), with 16 analog 12-bit inputs, two 12-bit DAC outputs, and a mix of digital I/Os. Iotech's "Personal Daqs" (www.iotech.com/catalog/daq/persdaq.html) are small sensors that offer up to 80 channels of analog and digital inputs, with a 22-bit A/D converter.

If you're building your own low-volume/lab-based link to a PC, and can't stomach the thought of designing your own USB hardware/software, consider the $79 USBSIMM card (usbsimm.home.att.net). This business card-sized controller uses an Anchor Chips 2131 (a USB-aware 8051 derivative). It's a neat and painless way to connect sensors or instruments to USB-based computers.

Compliance

The USB sponsoring organization has wisely created a compliance program to ensure that devices meet the standard's specifications. Though no law mandates that any device must pass these tests, doing so ensures that the user's experience with your products will be as trouble-free as possible. Products meeting the compliance program's requirements get added to the Integrator's List, a sort of imprimatur so customers can be sure that, from a communications perspective at least, the unit works properly.

Nothing stands still in this industry, not even the relatively new USB standard. Version 2.0, which may be formalized as you read this, extends the communications speed to a breathtaking 480Mbps. Clearly, new classes of applications are not far away.

Jack G. Ganssle is a lecturer and consultant on embedded development issues. He conducts seminars on embedded systems and helps companies with their embedded challenges. He founded two companies specializing in embedded systems. His "Break Points" column appears monthly in this magazine. Contact him at jack@ganssle.com.

Resources

www.usb.org. Home of the USB organization, a consortium of members who promulgate and enhance the standard. The site has some useful developer information, especially their message forum

(www.usb.org/forums/developers/webboard.html), which stays busy with questions and answers from active developers.

Axelson, Jan. USB Complete. Madison, WI: LakeView Research, 1999.
This is a readable and comprehensive book that covers all aspects of actually building and coding USB devices. Also see Axelson's Web site (www.lvr.com/usb.htm). Her description of building a HID-class peripheral is the best around.

Garney, John, Ed Solari, Shelagh Callahan, Kosar Jaff, and Brad Hosler. USB Hardware & Software. San Diego, CA: Annabooks, 1998.
This book bills itself as "the definitive reference to the Universal Serial Bus," and so it is. A large, dense book, it covers about every nuance of USB communications. This is a "must have" for USB developers. The authors worked on the original USB specification.

Tan, Wooi Ming. Developing USB PC Peripherals. San Diego, CA: Annabooks, 1997.
This is a slender book that gives a good overview and includes some useful sample driver and firmware code on a diskette.